FR 000 #10
US

Europäisches Patentamt | European Patent Office | Office européen des brevets

Bescheinigung | Certificate | Attestation

Die angehefteten Unterlagen stimmen mit der ursprünglich eingereichten Fassung der auf dem nächsten Blatt bezeichneten europäischen Patentanmeldung überein.

The attached documents are exact copies of the European patent application described on the following page, as originally filed.

Les documents fixés à cette attestation sont conformes à la version initialement déposée de la demande de brevet européen spécifiée à la page suivante.

Patentanmeldung Nr. | Patent application No. | Demande de brevet n°

00402876.7

Der Präsident des Europäischen Patentamts;
Im Auftrag

For the President of the European Patent Office

Le Président de l'Office européen des brevets
p.o.

I.L.C. HATTEN-HECKMAN

DEN HAAG, DEN
THE HAGUE, 16/07/01
LA HAYE, LE

Eur päisches Patentamt | European Patent Office | Office eur péen des brevets

Blatt 2 der Besch inigung
Sheet 2 of the certificate
Page 2 de l'attestation

Anmeldung Nr.:
Application no.: 00402876.7
Demande n°:

Anmeldetag:
Date of filing: 17/10/00
Date de dépôt:

Anmelder:
Applicant(s):
Demandeur(s):
Koninklijke Philips Electronics N.V.
5621 BA Eindhoven
NETHERLANDS

Bezeichnung der Erfindung:
Title of the invention:
Titre de l'invention:
Binary format for MPEG-7 instances

In Anspruch genommene Priorität(en) / Priority(ies) claimed / Priorité(s) revendiquée(s)

Staat: State: Pays:	Tag: Date: Date:	Aktenzeichen: File no. Numéro de dépôt:

Internationale Patentklassifikation:
International Patent classification:
Classification internationale des brevets:
/

Am Anmeldetag benannte Vertragstaaten:
Contracting states designated at date of filing: AT/BE/CH/CY/DE/DK/ES/FI/FR/GB/GR/IE/IT/LI/LU/MC/NL/PT/SE/TR
Etats contractants désignés lors du depôt:

Bemerkungen:
Remarks:
Remarques:

DESCRIPTION

The present invention concerns a MPEG-7 like transmission systems having a transmitter for transmitting encoded instances of MPEG-7 like description elements, and a receiver for receiving and decoding said encoded instances.
It also concerns a transmitter and a receiver for use in such a transmission system.
It further concerns a signal transporting such encoded instances, and an intermediate format information intended to be used in such a transmitter or in such a receiver.

The invention is described in the context of MPEG-7. But it is applicable to any system using XML-schemas

For a general description of what is MPEG-7 it may be reported to the article "Everything you wanted to know about MPEG-7:part 2" written by frank Nack and Adam T. Lindsay and published in the IEEE Multimedia October-December 1999.

2. Proposal overview

This proposal presents a binary format for MPEG-7 instances based on the serialisation of description fragments being composed of a structuring key and a *description element* value. Each *description element* (either an element or an attribute in XML) is represented by an independent fragment in the bitstream, ensuring random-access to elements and attributes as well as a high level of flexibility as far as the incremental transfer is concerned. Our fragment approach also takes into account the fundamental flexible and extensible nature of MPEG-7 by using the schema expressed in DDL to compute the structuring key of a given description element. Moreover, the coding scheme and binary syntax of each fragment has been designed in a way that will allow an easy integration with the future MPEG-7 instance update protocol, even though the update of the instances is out of the scope of this proposal. An overview of the proposal and the way MPEG-7 instances are encoded and decoded is given in figure 1.

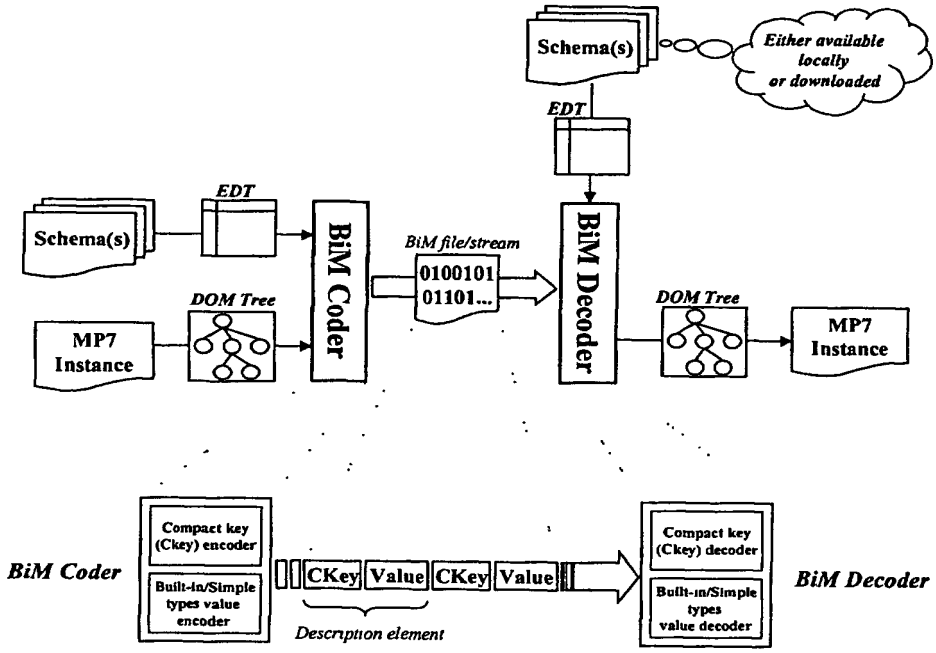


Figure 1 : BiM coding/decoding overview

The fragment approach allows the proposed binary format to achieve the following functionalities :

- Random access to instance elements & attributes
- Incremental non-ordered and scalable transfer.
- Compactness : only elements and attributes that have a content are coded.
- Easy integration with instance update protocol.
- Easy parsing and partial instantiation of binary MPEG7 descriptions.

The other advantages of the proposed format are captured by the use of an intermediate representation of the schema. Indeed, the *Element Declaration Table* (see section 3.1), directly and unambiguously generated from the schema, allows to share a common knowledge about the possible valid instances between the server and the client, in a form dedicated to the binary encoding and decoding of these instances. This common knowledge, gathering information such as structure, type, and tag name of the elements and attributes, does not need to be sent to the client, which leads to an efficient schema-aware encoding of the instances. This allows also the binary format to achieve a full extensibility support for future schemas defined inside or outside MPEG-7.

3. MPEG-7 Instance encoding

This section describes the instance encoding process and defines the binary syntax and semantics of the independent fragments of the bitstream. As mentioned in section 2, each *description element* (either an element or an attribute in the XML terminology) that has a primitive type content (e.g. built-in type, simple type, a descriptor with its own binary representation) will be encoded as an independent fragment composed of a structuring key and a content value. However, the elements within the XML hierarchy being only containers (no real content) will not be transmitted but inferred at the decoder side from the Element Declaration Table.

The EDT is thus used :

- at the encoder side in order to compute the structuring key of each description element
- at the decoder side in order to find back :
  - ✓ all the missing structural elements.
  - ✓ the description element nature (element/attribute).
  - ✓ the description element name.
  - ✓ the description element type in order to decode the value.

3.1 Element Declaration Table

3.1.1 Principles & definitions

The Element Declaration Table (in short EDT) is primarily intended to contain all the information needed to encode and decode any MPEG-7 instance that is valid with respect to a given schema definition. One can see the EDT as an exhaustive definition of the possible valid instances, generated uniquely and unambiguously from the schema by developing the element and attribute declaration structures. Indeed, the XML-schema (or DDL) gives mainly two kinds of information : On the one hand, the location of all the possible elements and attributes within the XML instance hierarchy is specified by means of complex type definitions (either named or anonymous) and element declarations. On the other hand, the type of their value is given through the use of built-in datatypes and simple type definitions. For each element or attribute that is specified in the schema and that can be found in the instance, the Element Declaration Table is gathering its name (e.g. the tag name for an element), its type, its nature (element or attribute) and a key specifying unambiguously its location within the hierarchical XML structure. While the schema is defining what an instance should look like for validation and interoperability purpose, the EDT is stating what an instance will look like from a structural perspective for coding purpose.

The basics of the Element Declaration Table and its use in the encoding and decoding process stand in the so-called *structuring key*, intended to uniquely identify :

- the type and name of the description element being transmitted.
- its location in the tree structure.

The syntax of this structuring key is a dotted notation where the dots denote hierarchy levels and the numbering at each level is performed by expanding all the elements and attributes declarations from the schema.
Let us take a simple example to illustrate the structuring key numbering syntax.
Consider the following schema :

```
<complexType name="complexType1">
   <sequence>
      <element name="Element1" type="type1"/>
      <element name="Element2" type="type2" minOccurs="0" maxOccurs="unbounded"/>
      <element name="Element3">
         <complexType>
            <sequence>
               <element name="Element4" type="type4" minOccurs="0" maxOccurs="1"/>
               <element name="Element1" type="type1"/>
            </sequence>
         </complexType>
      </element>
   </sequence>
   <attribute name="Attribute1" type="type4"/>
</complexType>
<element name="GlobalElement" type="complexType1"/>
```

The EDT, seen as a development of all schema element declarations, would contain among other information the following element names together with their corresponding structuring key :

Name	Structuring key	(...)
GlobalElement	0	
Element1	0.0	
Element2	0.1[]	
Element3	0.2	
Element4	0.2.0	
Element1	0.2.1	
Attribute1	0.3	

Note that the brackets in the *Element2* structuring key denote the presence of a multiple occurrence element. Moreover, *Element2 and Element4* are taken into account in the numbering even though they are optional elements. This allows to state a unique and unambiguous numbering of all the possible elements that can appear in any instance, disregarding the presence of a given element in a given instance. The resulting key is thus common to any instance that is valid with respect to the original schema. Note also that the *Element1* appears twice in the EDT since it can be instantiated at different locations within the tree structure.

The process of extracting the EDT records and theirs structuring key is comparable to develop all the element declarations in order to come up with an XML structure (e.g. DOM) of the biggest instance (the one instantiating all possible elements and attributes) corresponding to a given schema. Nevertheless, this "biggest" instance is infinite as soon as the schema defines self-embedding structures, commonly used within the MPEG-7 community to represent hierarchical description structures (e.g. SegmentDS can contain other Segment DSs). Hence, there is a clear need for capturing the self-containment in the EDT. This is done by specifying, in case of a self-contained element, its ancestor in the tree structure that has the same complex type. Such an element is thus not expanded further in the EDT.

Let us illustrate the latter concept of self-containment with a new example :
Consider the following schema :

```
<complexType name="complexType1">
   <sequence>
      <element name="Element1" type="complexType2"/>
      <element name="Element2" type="type2" minOccurs="0" maxOccurs="unbounded"/>
      <element name="Element3" type="type3"/>
   </sequence>
   <attribute name="Attribute1" type="type4"/>
</complexType>

<complexType name="complexType2">
   <sequence>
      <element name="Element4" type="type4"/>
      <element name="Element1" type="complexType2"/>
   </sequence>
</complexType>

<element name="GlobalElement" type="complexType1"/>
```

The EDT will contain, among other information such as the name and key of the elements, the self-containment field when relevant:

Name	Structuring key	Self-containment key	(...)
GlobalElement	0		
Element1	0.0 ←----------┐		
Element4	0.0.0 !		
Element1	0.0.1	0.0	
Element2	0.1[]		
Element3	0.2		
Attribute1	0.3		

At this stage, we have explained the syntax and the purpose of the structuring key in the element declaration table, allowing to identify an element name, type and location within the instance structure with a single notation. We have now to make the distinction between the structuring key found in the EDT and the structuring

key that will be encoded as a description element identifier in an instance binary fragment. The latter one can be seen as an instantiation of the former one. Indeed, the multiple occurrence elements will be actually indexed (resulting in keys such as 0.1[0], 0.1[1], ...) and the self-containment loops will be developed (resulting in keys such as 0.0.1.1.0 that do not appear in the EDT but can be derived from it).

In case of non-self-contained hierarchies, the mapping between the EDT structuring key and the instance structuring key is straightforward. Indeed, one has simply to remove the indexes found in the instance element to find back the EDT record that corresponds to the said element. For instance, the element represented by the key 0.1[5] is an *Element2*, the fifth being present in *globalElement*.

In case of self-contained hierarchies, the mapping is a bit more tricky since the instance structuring key does not explicitly appear in the EDT. The algorithm to find back the EDT record from the instance structuring key is given hereunder using C-like code :

Algorithm (1) :
Let `instance_key` be the instance structuring key of a given description element.
Let `edt_key` be the corresponding key as found in the EDT
Let `prefix(key)` be the largest prefix (*n* first tokens) of `key` that actually exists in the EDT.
Let `suffix(key)` be the last tokens of `key` so that `key = prefix(key)+suffix(key)`.
Let `self_cont(key)` be the self-containment key.

```
while ( prefix(instance_key) != instance_key )
{
   instance_key = self_cont( prefix( instance_key ) ) + suffix( instance_key );
}
edt_key = instance_key ;
```

Example : The instance contains the following structuring key : 0.0.1.1.0
Applying step by step the algorithm described above gives :
`instance_key` = 0.0.1.1.0
prefix(instance_key) = 0.0.1
`instance_key` = *self_cont(prefix(instance_key))* + `suffix( instance_key )` = 0.0 + 1.0 = 0.0.1.0
prefix(instance_key) = 0.0.1
`instance_key` = *self_cont(prefix(instance_key))* + `suffix( instance_key )` = 0.0 + 0 = 0.0.0
Which leads finally to :
`edt_key` = 0.0.0

Let us now summarise which fields should the Element Declaration Table contain in order to fulfil our requirements (a knowledge shared between the server and the client, generated from the schema and suited for encoding and decoding of description elements). Remember that this table is not sent to the client and can be computed only once per schema. An complete example of EDT can be found in Annex 1.

Field	Syntax	Semantics
Name	String	The name of the description element (e.g. tag name)
Type	String	The type of the description element. In case of a complex type, the type name as found in the schema. In case of a simple type, the name of the built-in type from which it derives.
Structuring key	Dotted notation	The unique key that specifies the location of the description element in the tree structure.
Self-containment key	Dotted notation	In case of a self-contained hierarchy, the key corresponding to the ancestor of the element that has the same complex type.
Elem/attr flag	Boolean	Is the description element an element or an attribute ?
Indexed flag	Boolean	Is the description element indexed (maxOccurs>1) ?
BiM primitive type	Boolean	Is the description element a primitive (i.e. built-in/simple) type or a complex type (see section 3.2.4.2) ?
List flag	Boolean	Is the description element type a derivation by list ?

3.1.2 Handling namespaces

In the EDT definitions and principles listed in the previous section, the issue of namespaces is not dealt with. This is mainly because the use of namespaces and the way to identify them is still an unclear issue within MPEG-7. Nevertheless, it is to be noted that whatever the choice on schema and namespace identification is made, this will not affect fundamentally the Element Declaration Table and our encoding principles. Indeed, if a schema imports other schemas through the use of namespaces, they have to be downloaded and analysed during the EDT generation process and the development of all element declarations in order to draw an exhaustive list of records in the EDT. The name (or ID, URL see proposal M6142, section 2.1.) of the schema where the element has been declared can be easily added as an EDT field for future validation purpose. This information will thus not need to be sent out since it will be included in the EDT at the client/decoder side.

3.1.3 Compact structuring key

The *structuring key* as defined in 3.1.1 can be used to identify within the instance the description fragment being encoded. However, this key is very verbose since it contains as many tokens as hierarchy levels in the XML structure. We thus propose to use a more compact form of the *structuring key* , referred to as the *compact structuring key* (in short CSK). In the simpler case (no self-containment), the CSK is the structuring key EDT record number.

First, we need to add a key to the current list of EDT fields by numbering the EDT records. Applied on the example described in section 3.1.1, this would lead to :

Name	Compact Key	Structuring key	Self-containment key	(...)
GlobalElement	0	0		
Element1	1	0.0		
Element4	2	0.0.0		
Element1	3	0.0.1	0.0	
Element2	4	0.1[]		
Element3	5	0.2		
Attribute1	6	0.3		

An algorithm adapted from algorithm (1) is used to compute the CSK in the general case (with self-contained structures) from the instance structuring key :

Algorithm (2) :
Let `instance_key` be the instance structuring key of a given description element.
Let `cs_key` be the corresponding compact structuring key.
Let `prefix(key)` be the largest prefix (*n* first tokens) of `key` that actually exists in the EDT.
Let `suffix(key)` be the last tokens of `key` so that `key = prefix(key)+suffix(key)`.
Let `self_cont(key)` be the self-containment key.
Let `compact_form(key)` be the corresponding compact form of `key` in the EDT.

```
while ( prefix(instance_key) != instance_key )
{
   cs_key = cs_key + compact_form( prefix(instance_key) ) ;
   instance_key = self_cont( prefix( instance_key ) ) + suffix( instance_key );
}
cs_key = cs_key + compact_form( prefix(instance_key) ) ;
```

Example : We want to compute the CSK corresponding to the following structuring key : 0.0.1.1.0
Applying step by step the algorithm described above gives :
`instance_key` = 0.0.1.1.0
prefix(instance_key) = 0.0.1
`cs_key` = 3
`instance_key` = *self_cont(prefix(instance_key))* + `suffix( instance_key )` = 0.0 + 1.0 = 0.0.1.0
prefix(instance_key) = 0.0.1
cs_key = 3.3
`instance_key` = *self_cont(prefix(instance_key))* + `suffix( instance_key )` = 0.0 + 0 = 0.0.0

Which leads finally to :
cs_key = 3.3.2

Experiments have shown that such a compression of the structuring key leads to a significant gain regarding the size of the key while offering exactly the same functionality. The way to decode the CSK is given in 4.2. Note that in the above example, the element is not a multiple occurrence element for sake of expression simplicity. It is nevertheless to be noted that each token of the instance structuring key (resp. the instance CSK) might be indexed (resp. contain several indexes). A more complete example can be found in annex 1.

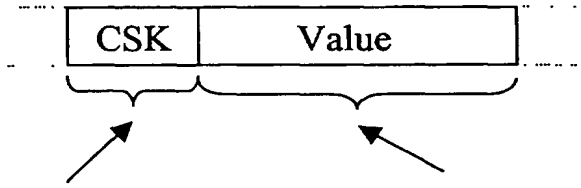
3.2 Binary syntax

3.2.1 File header

The header of the file should at least contain the schema identification (either an MPEG-defined ID or a URL as proposed in M6142). The binary syntax of the header is not specified further in this proposal.

3.2.2 Description fragment

Each description fragment is composed of a compact structuring key and a description element value, as described hereunder :



Compact Structuring Key
The size in bytes is variable

Content value (e.g. float, string, descriptor)
The size in bytes is variable but is stated in the packet header

The binary syntax of the CSK and the description element value is defined in the following sections.

3.2.3 Compact structuring key

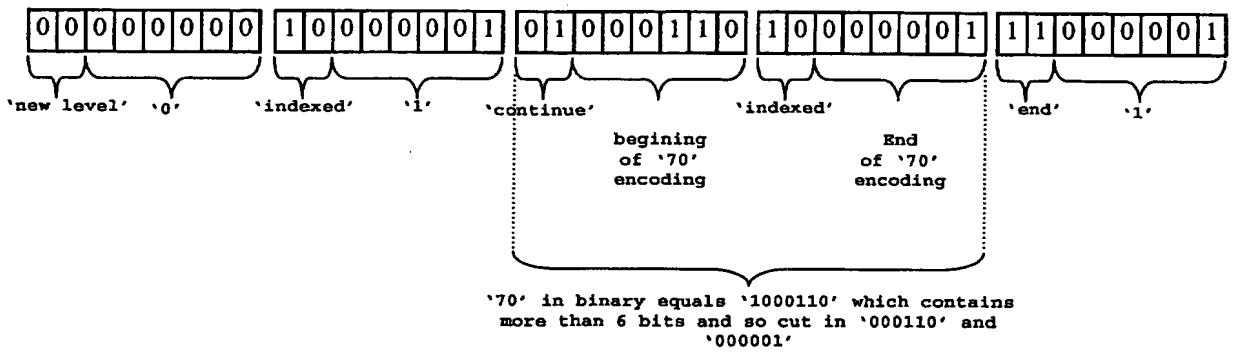
The generic form of the compact structuring key is as follows :

Token

Key[*ind*][*ind*](...)[*ind*].*Key*[*ind*][*ind*](...)[*ind*].(...) , where all keys and indexes are integer values coded using a variable number of bytes. The whole structuring key is thus coded using a variable set of bytes, each of them being controlled by the 2 most significant bits with the following semantics :

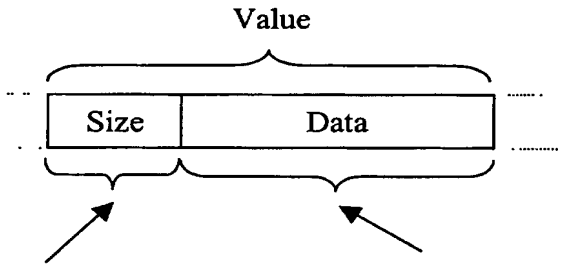
Control bits		Semantics
Bit7	Bit6	
0	0	*"New level"* : The next byte represents the beginning of a new token.
0	1	*"Continues"* : The next byte is to be interpreted as the following bits of the current key or index
1	0	*"Indexed"* : The next byte is the beginning of the next index within the current token.
1	1	*"End"* : The current byte is the last byte of the structuring key.

Example : binary encoding of the compact key « 0.1[70][1] » is :



3.2.4 Description Element Value

3.2.4.1 Data size encoding



Size of the data packet
The size is coded using a variable amount of bytes

Content value (e.g. float, string, descriptor)
The size in bytes of the data is given in the header

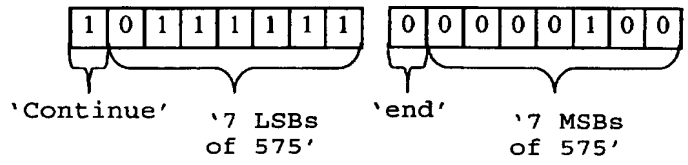
Before adding a data value to the binary file or stream, the size in bytes of the data block is coded. This aims at informing the decoder about the size of data to be decoded and guaranties an easy random access to data and fast stream parsing. Since certain primitive data types can imply large amounts of bytes (e.g. think of free text annotation or movie scripts), we propose to code the data size using a variable number of bytes to avoid the statement of limitations within the standard.

The length is thus coded by default using one byte, with the most significant bit being interpreted as follows :

Bit7	Semantics
0	"end" : The length coding is finished
1	"continues" : The length coding continues on the next byte

Example :

Assume the size of a data to be coded is 575 (binary : 10 00111111), then :
The first byte is composed of the 7 less significant bits of the length value with the addition of a control bit specifying that another byte is required. The second byte contains the remaining bits with the "end" control bit.



3.2.4.2 Primitive types : support for basic and extended types

As already mentioned, a major advantage of the proposed coding scheme is to encode only the elements or attributes that contain a value, and skip the elements that are only structural containers (e.g. with a complex type). This is allowed given that the structure can be inferred at the decoder side using the Element Declaration table.

Example :

Consider the following instance fragment (found in the core experiment test set) :

```
<GenericDS>
   <MediaInformation>
      <MediaProfile>
         <MediaInstance>
            <InstanceLocator>
               <MediaURL>imgs/img00587_add3.jpg</MediaURL>
            </InstanceLocator>
         </MediaInstance>
      </MediaProfile>
   </MediaInformation>
</GenericDS>
```

In this case, only the *MediaURL* would be encoded (as a string) using a structuring key that allows the decoder to reconstruct the whole structure from the EDT. The other container elements would not be transmitted.

In the general case, all the elements which type is known by the encoder and the decoder (i.e. for which a binary representation is available in a standard way and ensures interoperability) shall be encoded. We address here the issue of choosing the set of primitive types for which MPEG-7 should define a standard binary representation. Nevertheless, the scope of this document is the compression and the binary representation of the instance structure and is not to draw an extensive and exhaustive list of compression and/or optimal encoding schemes for every MPEG-7 built-in data types (we do not need to reinvent the wheel in an area that is widely developed in within other international standards).

The set of ***MPEG-7 basic types*** (for which a binary representation syntax and semantics must be defined) should include the XML-schema (or DDL) built-in types (e.g. string, float, ...) as well as some MPEG-7 specific basic types (e.g. unsignedInt1, unsignedInt2, ..., MediaTime, Matrix, ...). This set should then automatically include all the simple types that are derived from the above basic types by *restriction* (attribute *derivedBy="restriction"*) in the XML-schema sense. The latter assertion means for instance that there is no absolute need for defining a new binary representation for integers between 0 and 100 if a generic encoding scheme already exists for integers. In addition, the set should automatically include all the types that are derived from the above basic types by *list* (attribute *derivedBy="list"*). Indeed, the coding scheme of a list of "datatype" can be easily derived from the coding scheme of "datatype", appending individual elements within one chunk of data. The number of elements in the list can then be deduced from the chunk size and the size of a single element binary representation.

As a example (which is certainly not exhaustive), we used for our experiments the following set of basic types, together with the derived types of these (either by list or by restriction) :

- String
- Float
- Integer
- UnsignedInt8

We believe that it should be advantageous to augment the above mentioned list of basic types with some ***MPEG-7 extended types*** that might include complex types in the following cases :

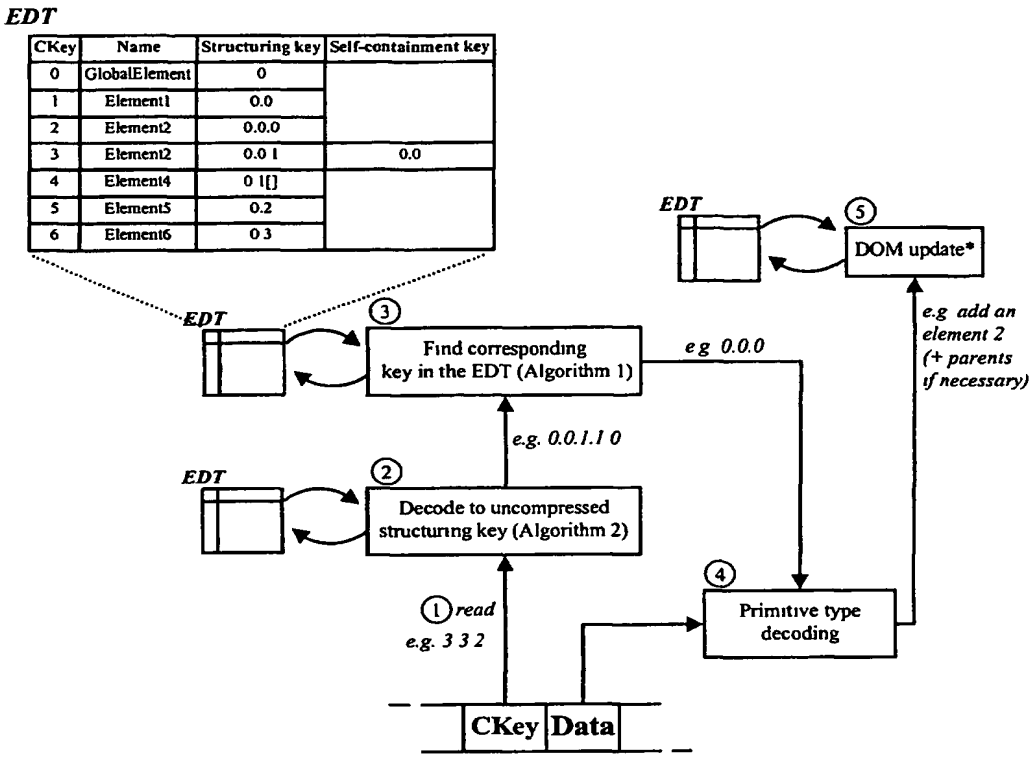
- There is no need for accessing randomly the embedded elements within the complex type structure.
- An efficient binary representation already exists

These criteria are certainly fulfilled in the case of descriptors as defined by the video and audio group. Indeed, a compact binary representation has already been defined and should be used and there is (most of the time) no need for accessing the individual parts of the descriptors (they make sense as a whole).

Note that there is probably a trade-off to find in this area since the efficiency (in terms of content compression) should increase with an increasing number of primitive types (which are encoded in an optimal way), but so does the complexity of the decoder which is supposed to include the decoding methods for all the standard primitive types.

4. MPEG-7 Instance decoding

The overall fragment decoding scheme is illustrated in figure 2. The following sections give detailed information on the parsing process, the compact structuring key decoding and the DOM update algorithm.

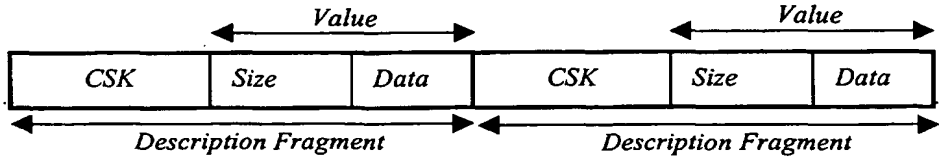


* This update concerns the current element as well as all its parents information.
That is why the *DOM update* not only needs the current element but also the complete structuring key.

Figure 2 : Binary description fragment decoding scheme

4.1 Parsing

Following our coding principles, we have seen that a BiM stream or a BiM file will contain only elements (resp. attributes) of the original instance that have a content value. The BIM stream parsing becomes easy since this stream is serialised with the following sequence :



The client can quickly have access to a particular data of the BiM stream according to its binary *Compact key*. The parsing is done by matching it with the current binary *Compact Key* in the stream and jumping from a *Description Fragment* to another using the data *Size* without decoding the current data. There is no need to use the EDT for parsing a BiM stream, access to EDT is only necessary for decoding purpose.

During the decoder processing, the current *Description Fragment* available in the stream is read. First, the binary *Compact Key* is read using controls bits embedded in the stream and decoded to its expanded form (structured key) using algorithm (3) as described in 4.2. Then, the current data *Size* is read and the corresponding data value can be accessed. With this information, the decoder can generate the corresponding part of the instance in memory as described in 4.3. A new *Description Fragment* is then read and the process is iterated until the stream is empty (resp. the end of file is reached).

4.2 Decoding the compact structuring key

The only purpose of the compact structuring key is to reduce the size of the BiM stream and is thus firstly decoded to its expanded form (structuring key) by the decoder before the general decoding phase. The algorithm that returns the structuring key corresponding to a compact one is given hereunder.

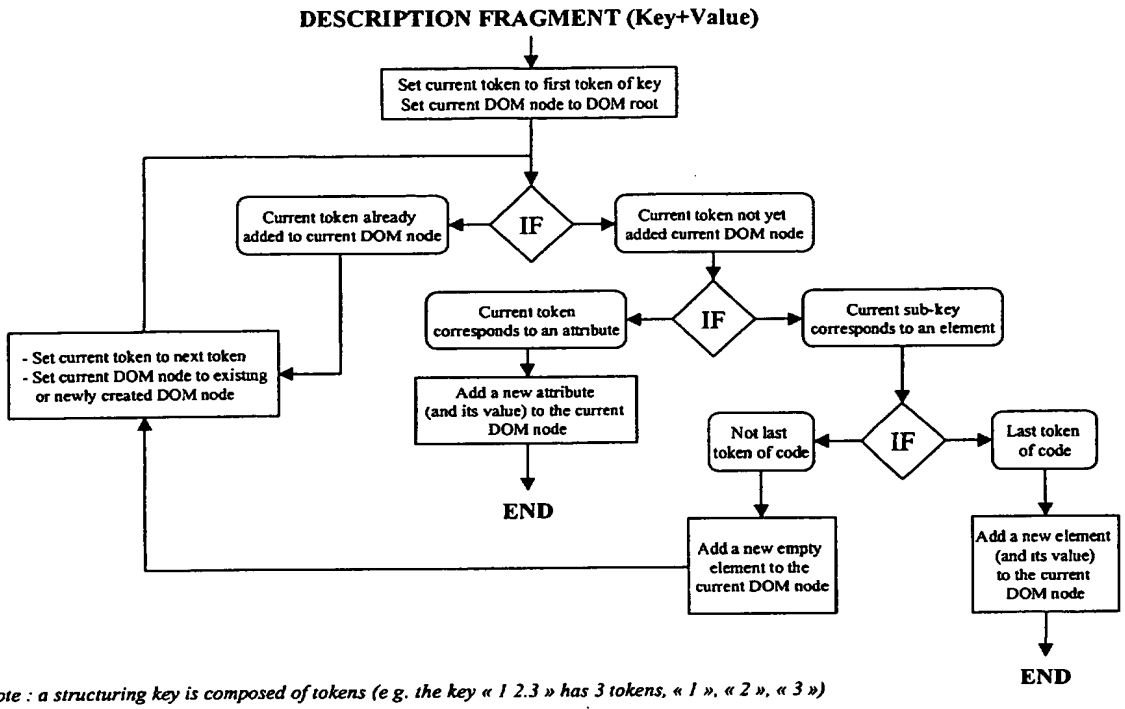
Algorithm (3) :
Let `resultNCKey` be the literal form of compact_key (result of the algorithm).
Let `compact_key` be the BiM instance compact structuring key of a given description element.
Let `current_key` be a token of the BiM instance compact structuring key `compact_key`.
Let `compact_key[i]` be the ith token of `compact_key`.
Let `size(compact_key)` be the number of tokens of `compact_key`.
Let `diffCode(key1, key2)` be sub-key obtained by removing the common sub-key of `key1` and `key2`
Let `NCKey(CKey)` be the corresponding literal form of the compact key `CKey`.
Let `self_cont(key)` be the self-containment key of `key`.

All indexes are first removed from `compact_key` and are put back at the end in the developed form of `compact_key`.

```
current_key = compact_key[0]
resultNCKey = NCKey(current_key)
for (i=1; i<size(compact_key); i++)
{
   previous_key = current_key ;
   current_key = compact_key[i] ;
   resultNCKey += "." + diffCode(NCKey(current_key), self_cont(previous_key)) ;
}
```

4.3 Updating the DOM

The details of the DOM update algorithm is given in figure 3. Note that this algorithm does not directly generate an MPEG-7 instance but a DOM (Document Object Model) tree, using DOM standard APIs from W3C that provide a hierarchical memory representation of an XML file as well as methods to work on it.



Note : a structuring key is composed of tokens (e g. the key « 1 2.3 » has 3 tokens, « 1 », « 2 », « 3 »)

Figure 3 : DOM update algorithm for a given description fragment

The decoder then parses and processes the next description fragment (structuring key, content) until the BiM file/stream is empty. Once the entire document is instantiated as a DOM tree, it is very easy to turn it into an XML file by recursively printing each DOM nodes.

5. Core Experiment results

5.1.1 Compression results

Our BiM encoder and decoder has been tested with the core experiment test set. This following table collects results of this tests.

Instance Name	Textual instance size (bytes)	Binary instance size (bytes)	Instance compression ratio	Textual structure size (bytes)	Binary structure size (bytes)	Structure compression ratio
CollectionStructureDS_1	1407947	238602	5,90	1236161	105158	**11,76**
CollectionStructureDS_2	8002	3641	2,20	4132	235	**17,58**
CollectionStructureDS_3	267322	93167	2,87	158812	5743	**27,65**
VideoSegmentDS_1	215997	56005	3,86	185456	30619	**6,06**
VideoSegmentDS_2	133229	32705	4,07	114653	16995	**6,75**
VideoSegmentDS_3	14531	2818	5,16	12635	1078	**11,72**
VideoSegmentDS_4	322868	84796	3,81	276912	46908	**5,90**
VideoSegmentDS_5	866416	223915	3,87	743075	122222	**6,08**
StillRegionDS_1	9207	2330	3,95	8099	1017	**7,96**
StillRegionDS_2	4075	980	4,16	3597	417	**8,63**
StillRegionDS_3	8647	2185	3,96	7617	953	**7,99**
VideoSegStillReg_1	13365	3394	3,94	10858	635	**17,10**
VideoSegStillReg_2	21612	5545	3,90	17830	1392	**12,81**
VideoSegStillReg_2	98619	26700	3,69	79400	5680	**13,98**

Average instance compression = 3.95 Average structure compression = 11.57

Compression ratios have been calculated in two cases : first case, the complete instance is taken into account (i.e. the entire XML instance file); second case, only instance structure is considered (i.e. tags and XML syntax without element/attribute data).
This separation is useful to quantify the BIM encoder efficiency. Indeed, we chose not to work on the compression of the content value (the data found between tags and within attributes) in the scope of this proposal. As explained in 3.2.4.2, the efficient encoding of the description element values using a standard set of primitive types would probably significantly increase the overall instance compression ratio).

In the following, we will hence concentrate on results related to the instance structure compression. Using our BiM encoding scheme, the average size of the core experiment instance is divided by about a factor of 12. But, as shown in the table, results may vary from an instance to another due to the significant structure differences between instances. A possible illustration of this fact is to compare structures of two instances that gives extreme compression ratios (i.e *VideoSegmentDS_4* and *CollectionStructureDS_3*).

In the *VideoSegmentDS_4* instance, this structure can be found :

```
<HistogramD HistogramNormFactor="1" NumberHistogramBins="64">
      <HistogramValue>529</HistogramValue>
      <HistogramValue>503</HistogramValue>
      <HistogramValue>843</HistogramValue>
      <HistogramValue>12355</HistogramValue>
      <HistogramValue>86</HistogramValue>
      <HistogramValue>199</HistogramValue>
      <HistogramValue>359</HistogramValue>
            .....
</HistogramD>
```

The HistogramD descriptor is developed with multiple occurrences of the HistogramValue element. In this case, the BIM encoder associates an independent compact key for each HistogramValue tag found in the instance which explains the relatively poor compression ratio of this structure compared to the average one. One can note that the random access to every single bin of the histogram is probably not needed and that designing the schema using the *derivedBy="list"* facet would have significantly improve the compactness of the description (both in the textual and in the binary domain).

Let us take a look to the *CollectionStructureDS_3* structure. Here is an fragment of the instance :

```
<GenericDS>
   <MediaInformation>
      <MediaProfile>
         <MediaInstance>
            <InstanceLocator>

   <MediaURL>../../inputdata/Collection/collection4.txt.dir/img00587_add3.jpg</MediaURL>
            </InstanceLocator>
         </MediaInstance>
      </MediaProfile>
   </MediaInformation>
</GenericDS>
```

In this case, only the tag MediaURL contains a data. Thus, only MediaURL will be coded to BIM stream with its associated data, which explains the good result found for this structure.

Add a conclusion sentence : 12 is a good compression ration considering our flexibility

5.1.2 Evaluation criteria

Answer to the evealution criteria given in the Core Experiment

5.1.3 Advanced functionalities

Give here the functionalities beyond the official requirements :

- Scalability (Put important fragment first, less useful descriptors at the end)
- Very easy integration in any instance update framework (simply add the update chunk or use the same decoding algorithm to update the DOM
- Mapping with the KLV encoding is straightforward. This might be of great help towards the integration of the 2 standards.

Annex 1 : Instance encoding example

In this annex, we give a complete example of en MPEG-7 instance encoding, starting from the schema, generating the EDT and finally giving an example of binary description fragment. The following schema is used to illustrate the encoding process. Note that this schema is relatively short but use several MPEG-7 principles that are relevant in our context for sake of completeness (use of indexed and self-contained elements).

SCHEMA EXAMPLE

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns:xsd="http://www.w3.org/1999/XMLSchema">

   <complexType name="MediaTimeType" content="elementOnly">
      <sequence>
         <element name="Start">
            <simpleType base="integer"/>
         </element>
         <element name="Stop">
            <simpleType base="integer"/>
         </element>
      </sequence>
      <attribute name="timeunit" type="string" use="required"/>
   </complexType>

   <complexType name="VideoSegmentType" content="elementOnly">
      <sequence>
         <element name="keyFrame" minOccurs="1" maxOccurs="unbounded">
            <simpleType base="string"/>
         </element>
         <element name="Annotation" type="string" minOccurs="0" maxOccurs="1"/>
         <element name="MediaTime" type="MediaTimeType" minOccurs="0" maxOccurs="1"/>
         <element name="VideoSegment" type="VideoSegmentType" minOccurs="0" maxOccurs="unbounded"/>
      </sequence>
      <attribute name="id" use="required">
         <simpleType base="string"/>
      </attribute>
   </complexType>

   <element name="VideoSegment" type="VideoSegmentType"/>

</schema>
```

ELEMENT DECLARATION TABLE EXAMPLE

For the schema given above, an Element Declaration Table (EDT) is created. This is a simplified representation of the schema that contains the minimum information needed to code whatever instance based on this schema. The EDT is generated once and available for coding and decoding an instance that refers to this schema.

COMPACT KEY	STRUCTURING KEY	SELF-CONTAINMENT KEY	NAME	TYPE NAME	FLAGS			
0	0		VideoSegment	VideoSegmentType	Element	Container	Indexed : no	List : no
1	0.0[]		KeyFrame	string	Element	Primitive	Indexed : yes	List : no
2	0.1		Annotation	AnnotationType	Element	Primitive	Indexed : no	List : no
3	0.2		MediaTime	MediaTimeType	Element	Container	Indexed : no	List : no
4	0.2.0		Start	integer	Element	Primitive	Indexed : no	List : no
5	0.2.1		Stop	integer	Element	Primitive	Indexed : no	List : no
6	0.2.2		timeunit	string	Attribute	Primitive	Indexed : no	List : no
7	0.3[]	0.0	VideoSegment	VideoSegmentType	Element	Container	Indexed : yes	List : no
8	0.4		id	string	Attribute	Primitive	Indexed : no	List : no

For more information about :

- *Compact Key*, see sections 3.1.3, 3.2.3, 4.2
- *Structuring Key*, see section 3.1.1
- *Self-containment Key*, see section 3.1.1
- *EDT principles*, see section 3.1.1

INSTANCE EXAMPLE

The following instance is an valid instance example with respect to the schema given above. On the left of are given the Structuring Key and associated Compact Key of the element defined in the corresponding line. On the right, the same information is given for attributes. Only bold-character Compact Keys are used in the BIM code of the instance (primitive types).

Element Structuring Key	Element Compact Key	Instance	Attribute Structuring Key	Attribute Compact Key
		<?xml version="1.0" encoding="UTF-8"?>		
0	0	<VideoSegment id="VS1">	0.4	**8**
0.0[0]	**1[0]**	<keyFrame>./../video/Scotland.jpg</keyFrame>		
0.1	**2**	<Annotation>My trip in Scotland</Annotation>		
0.2	3	<MediaTime timeunit="PT1N30F">	0.2.2	**6**
0.2.0	**4**	<Start>0</Start>		
0.2.1	**5**	<Stop>1500</Stop>		
		</MediaTime>		
0.3[0]	7[0]	<VideoSegment id="VS2">	0.3[0].4	**7[0].8**
0.3[0].0[0]	**7[0].1[0]**	<keyFrame>./../video/video_landscape/landscape1.jpg</keyFrame>		
0.3[0].0[1]	**7[0].1[1]**	<keyFrame>./../video/video_landscape/landscape2.jpg</keyFrame>		
0.3[0].0[2]	**7[0].1[2]**	<keyFrame>./../video/video_lanscape/landscape2.jpg</keyFrame>		
0.3[0].3[0]	7[0].7[0]	<VideoSegment id="VS3">	0.3[0].3[0].4	**7[0].7[0].8**
0.3[0].3[0].0[0]	**7[0].7[0].1[0]**	<keyFrame>./../video/video_landscape/forest.jpg</keyFrame>		
0.3[0].3[0].1	**7[0].7[0].2**	<Annotation>forest of oaks</Annotation>		
0.3[0].3[0].2	7[0].7[0].3	<MediaTime timeunit="PT1N30F">	0.3[0].3[0].2.2	**7[0].7[0].6**
0.3[0].3[0].2.0	**7[0].7[0].4**	<Start>0</Start>		
0.3[0].3[0].2.1	**7[0].7[0].5**	<Stop>200</Stop>		
		</MediaTime>		
		</VideoSegment>		
0.3[0].3[1]	7[0].7[1]	<VideoSegment id="VS4">	0.3[0].3[1].4	**7[0].7[1].8**
0.3[0].3[1].0[0]	**7[0].7[1].1[0]**	<keyFrame>./../video/video_landscape/beach.jpg</keyFrame>		
0.3[0].3[1].1	**7[0].7[1].2**	<Annotation>The north beach</Annotation>		
0.3[0].3[1].2	7[0].7[1].3	<MediaTime timeunit="PT1N30F">	0.3[0].3[1].2.2	**7[0].7[1].6**
0.3[0].3[1].2.0	**7[0].7[1].4**	<Start>200</Start>		
0.3[0].3[1].2.1	**7[0].7[1].5**	<Stop>450</Stop>		
		</MediaTime>		
		</VideoSegment>		
		</VideoSegment>		
		</VideoSegment>		

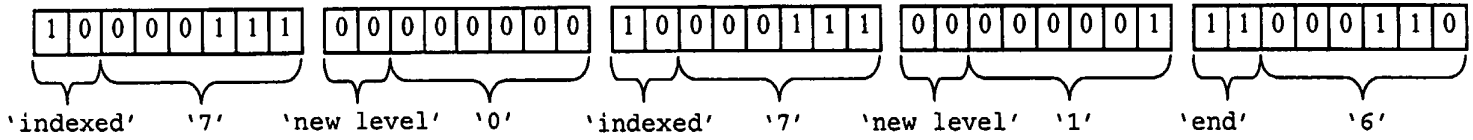
BIM DESCRIPTION FRAGMENT EXAMPLE

In order to encode a description fragment from the instance given above, a Compact Structuring Key (bold characters in the EDT) as defined in 3.2.3 is first encoded followed by the description fragment value (see section 3.2 for the binary syntax), encoded with the appropriate MPEG-7 primitive type binary representation. For example, here is a part of the instance (The media time of the video segment VS4) that will become a binary description fragment :

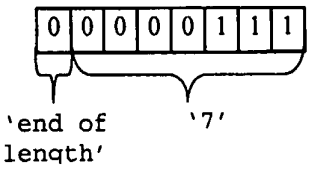
(...) <MediaTime timeunit="PT1N30F"> (...)

Here, as explained in 3.2.4.2, only the value of the attribute *timeunit* is coded since the element *MediaTime* has no primitive type content.

compact key : 7[0].7[1].6



Description value length in bytes : 7



string data: PT1N30F

Description fragment value (here a string) coding is performed by converting string characters to bytes using usual character coding.

CLAIMS

1. A MPEG-7 like transmission system having a transmitter for transmitting encoded instances of MPEG-7 like description elements, and a receiver for receiving and decoding said encoded instances,
said transmitter and said receiver both having:
- one or more stored schemas which notably contain definitions (called types) of hierarchical structures of description elements,
- means for deriving from said schemas, intermediate format information which contain, for each description element that is specified in the schemas, at least the name, type, nature (element or attribute), and location within the hierarchical structure, an indication of the ancestor of the description element in case of self-contained hierarchy, and an indication of the existence of multiple occurrence of the description element,
said transmitter further having:
- means for encoding the description elements of said instances which have a content, as fragment composed of an instance structuring key and a content value, said instance structuring key being derived from said intermediate format information;
and said receiver further having:
- means for decoding the received encoded instances by using said intermediate format information in order to retrieve the name, nature and type of the encoded description elements, and to retrieve the description elements which were not transmitted because they had no content.
2. A MPEG-7 like transmission system as claimed in claim 1, wherein said instance structuring key is obtained from an instantiation of said location information wherein multiple occurrence description elements are indexed and self-containment loop are developed.
3. A MPEG-7 like transmission system as claimed in claim 1, wherein said intermediate format information are stored in a table called "Element Declaration Table" having the following fields and flags for each description element:
- a name field indicating the of the description element,
- a type field indicating the type of the description element,
- a structuring key field specifying the location of the description element in the tree structure,
- a self-containment key field used in case of self-contained hierarchy for indicating the ancestor of the description element that has the same type
- an element / attribute flag to indicate whether the description element is an element or an attribute,
- an index flag to indicate if there are multiple occurrence of the description element,
- a BiM built in flag indicating whether the description element is a built-in/simple type or a complex type,
- a list flag indicating whether the description element type is a derivation by list.
4. A MPEG-7 like transmission system as claimed in claim 2, wherein said instanced key is a compression of said instantiation.
5. A MPEG-7 like transmission system as claimed in claim 2, wherein said instance structuring key consists of one or more token, each token comprising one key and one or more index, and is coded by using a variable set of bytes each of them having control bits to indicate whether it is the last byte of the instance structuring key or whether the next byte is the beginning of a new token, or contains the following bits of the current key or index of the current token, or is the beginning of the next index of the current token.

6. A MPEG-7 like transmission system as claimed in claim 1, wherein said content value comprises a data field and a data length field, each byte of the data length field having a specific bit indicating if this byte is the last byte of the data length field.
7. A transmitter intended for use in a MPEG-7 like transmission system as claimed in one of claims 1 to 6.
8. A receiver intended for use in a MPEG-7 like transmission system as claimed in one of claims 1 to 6 .
9. A signal transporting independent fragments composed of an instance structuring key and a content value where the instance structuring key is a compression form of an instantiation of the location of e description element in a hierarchical structure, in which multiple occurrence description elements are indexed and self-containement loop are developed.
10. A signal as claimed in claim 9, wherein each instance structuring key which consists of one or more token, each token comprising one key and one or more index, is coded by using a variable set of bytes each of them having control bits to indicate whether it is the last byte of the instance structuring key or whether the next byte is the beginning of a new token, or contains the following bits of the current key or index of the current token, or is the beginning of the next index of the current token.
11. A computer program for use in a transmitter as claimed in claim 7 in order to encode said description element.
12. A computer program for use in a receiver as claimed in claim 8 in order to decode said encoded instances.
13. An element declaration table intended to be used in a transmitter as claimed in claim 7 or in a receiver as claimed in claim 8 and having the following fields and flags for each description element:
    - a name field indicating the of the description element,
    - a type field indicating the type of the description element,
    - a structuring key field specifying the location of the description element in the tree structure,
    - a self-containment key field used in case of self-contained hierarchy for indicating the ancestor of the description element that has the same type
    - an element / attribute flag to indicate whether the description element is an element or an attribute,
    - an index flag to indicate if there are multiple occurrence of the description element,
    - a BiM built in flag indicating whether the description element is a built-in/simple type or a complex type,
    - a list flag indicating whether the description element type is a derivation by list.